ilmedia

TECHNISCHE UNIVERSITÄT ILMENAU

Marion Braunschweig, Mathias Weiß

Computergrafik - die Realisierung der Mausinteraktionen Panning, Zoom und Rotation in einem OpenGL - Windows - Programm

Technische Universität Ilmenau
Ilmenau 2010

URN: urn:nbn:de:gbv:ilm1-2010200144

Computergrafik - die Realisierung der Mausinteraktionen Panning, Zoom und Rotation in einem OpenGL - Windows - Programm

Marion Braunschweig, Mathias Weiß: Technische Universität Ilmenau

1. Einleitung

Wissenschaftliche Visualisierungen nutzen in hohem Maße den universellen Grafikstandard OpenGL, mit dem realistische 3D-Darstellungen unabhängig vom verwendeten Rechner und unabhängig vom verwendeten Betriebssystem programmiert werden können /2/. Sind die 3D-Objekte in einem Fenster auf dem Bildschirm erzeugt, ist es wichtig, diese Objekte durch Anklicken mit der Maus auch verschieben zu können (Panning, kurz Pan), durch Nutzung des Mausrades vergrößern und verkleinern zu können (Zoom) und natürlich mit der Maus auch Drehen zu können (Rotation). Der Artikel beschreibt eine Softwarelösung für Windows, basierend auf der Sprache C# und unter Nutzung der Microsoft .NET Technologien. Die OpenGL-Funktionalität wird durch die Einbindung des Tao-Frameworks /4/ erreicht. Die zugrundliegende Mathematik /1/ und /6/ wird ebenfalls beschrieben. Vorausgesetzt werden grundlegende Kenntnisse der Programmiersprache C und Grundkenntnisse von OpenGL.

2. Grundprinzipien

OpenGL zeichnet 3D-Objekte unter Nutzung von kartesischen Koordinaten (x, y, z) in einem Darstellungsraum (viewing frustum), der durch sechs Clipping-Ebenen begrenzt wird. Im Bild 1 befindet sich der Betrachter im Koordinatenursprung und der Darstellungsraum ist mit den beispielhaften Größen:
near , far, Öffnungswinkel fovy und Seitenverhältnis (Breite/Höhe) w/h
festgelegt worden.

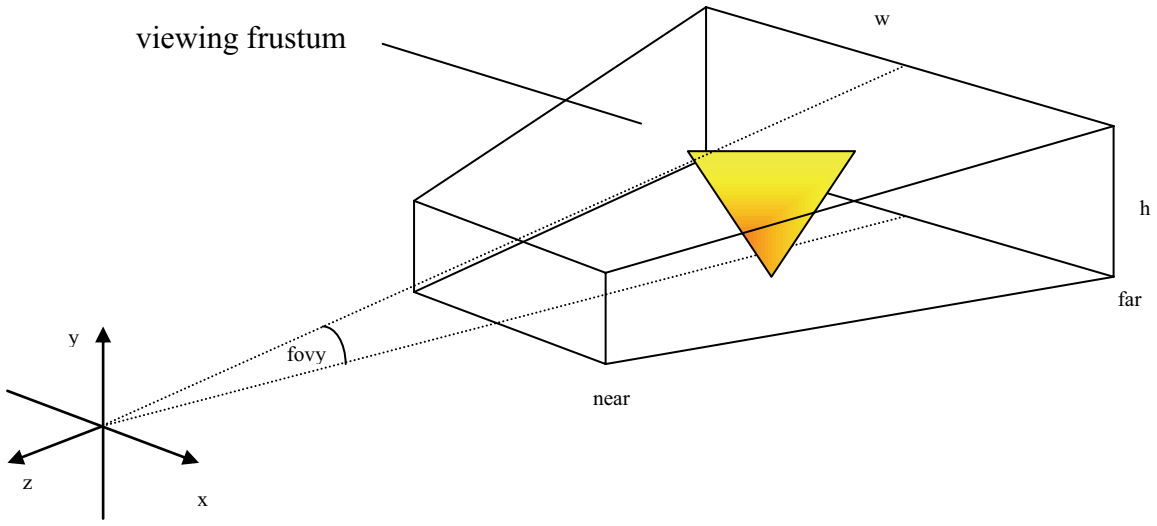

Bild 1: Der Darstellungsraum in OpenGL

Die Größe dieses Darstellungsraumes verwaltet OpenGL in einer 4x4-Matrix, die den Namen Projection-Matrix trägt. Eine Möglichkeit zur Initialisierung dieser Matrix bietet der Funktionsaufruf:
gluPerspective(double fovy, double w/h, double near, double far);
mit den oben eingeführten Größen.
Das eigentliche Zeichnen erfolgt durch die Nutzung der grafischen Primitive (drawing primitives), die OpenGL bietet und erfordert immer die Angabe konkreter Punkte im Raum. Ein Punkt im Raum ist ein Vertex (Plural: vertices), der in homogenen Koordinaten (x,y,z,w) angegeben wird (in allen folgenden Betrachtungen ist w=1.0) siehe Bild 2.

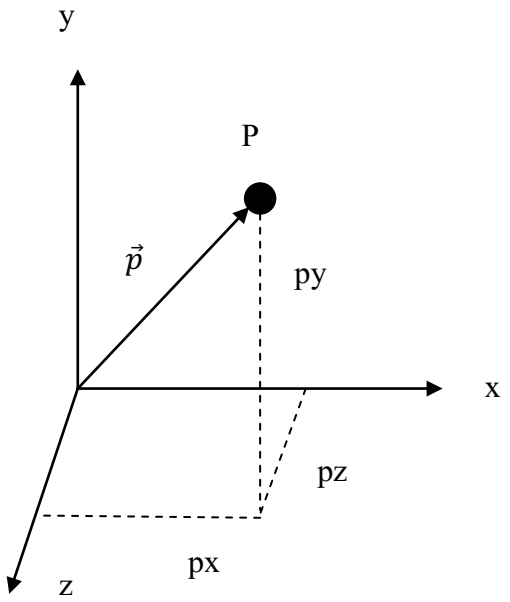

Bild 2: Ein Bildpunkt P

Die homogenen Koordinaten des Punktes P - diese sind die Weltkoordinaten - sind zu programmieren. Vorteil: Der Programmierer muss nicht in Bildschirmpixeln denken.
Der Vektor $\vec{p}$ lautet:

$$\vec{p} = \begin{bmatrix} px \\ py \\ pz \\ w \end{bmatrix}$$

Der Programmierer muss also immer die kartesischen Koordinaten px, py, pz genau angeben, um ein Objekt zu zeichnen. Um diese Arbeit zu erleichtern, besteht in OpenGL auch die Möglichkeit, das Koordinatensystem zu transformieren. Die zugehörigen Funktionsaufrufe lauten:
glTranslated (double x, double y, double z);
verschiebt das Koordinatensystem um den Vektor x, y, z.
glRotated (double alpha, double x, double y, double z);

rotiert das Koordinatensystem um den Winkel alpha, wobei die Drehachse durch den Vektor mit den Koordinaten x, y, z vorgegeben wird.
glScaled(double Sx, double Sy, double Sz);
skaliert das Koordinatensystem (d.h. dehnen, stauchen, spiegeln), wobei jede Achse mit dem entsprechenden Faktor Sx, Sy, Sz multipliziert wird.
Die Ergebnisse aller Transformationen speichert OpenGL ebenfalls in einer 4x4-Matrix; diese Matrix heißt **Modelview-Matrix MV** und spielt bei allen nachfolgenden Betrachtungen eine zentrale Rolle.

3. OpenGL-Pipeline

Nachdem der Programmierer die Vertices festgelegt hat, durchlaufen die homogenen Weltkoordinaten eine Reihe von Transformationen mit dem Ziel, diejenigen Bildschirmpixel zu berechnen, die auf einem 2D-Display der 3D-Darstellung der Objekte in Weltkoordinaten entsprechen. Diese Reihe von Transformationen heißt OpenGL-Pipeline. Wenn man sich auf die reinen geometrischen Transformationen, die in der Modelview-Matrix MV stehen, beschränkt, erhält man folgendes Prinzip nach Bild 3.

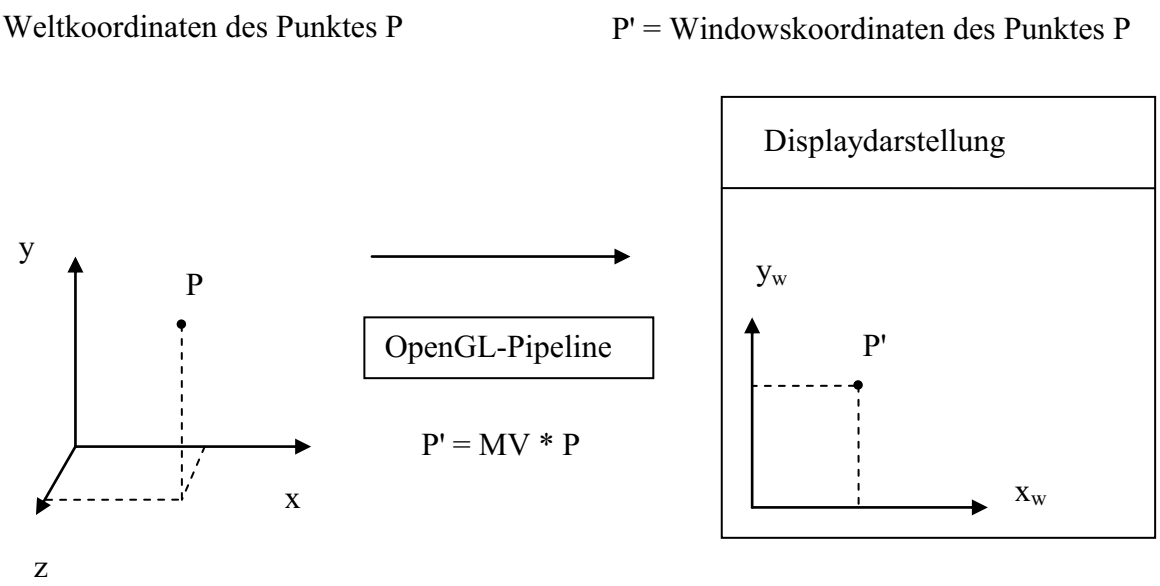

Bild 3: OpenGL-Pipeline

Die Displaydarstellung ist selbstverständlich nur zweidimensional (Windowskoordinaten x_w, y_w) - rechnerintern verwaltet OpenGL aber zu jedem Pixel noch den dazugehörenden z-Wert z_w in einem sogenannten z-Buffer, einem Tiefenpuffer.
Die Windowskoordinaten erhält man durch die Gleichung

$P' = MV * P$ mit MV = Modelview-Matrix.

Natürlich kann man die Gleichung auch umstellen zu

$P = MV^{-1} * P'$ mit MV^{-1} = inverse Modelview-Matrix.

Diese Gleichung werden wir für die Durchführung der Rotation als Mausinteraktion benötigen.

4. Programmstruktur und Programmablauf

Die einfachste Programmstruktur zeichnet das geforderte Bild in der Funktion drawScene() - dann befindet sich das Betriebssystem Windows in einer Warteschleife. Nutzeraktionen (Maus, Tastatur) lösen entsprechende Windows-Event-Funktionen aus - siehe Bild 4.

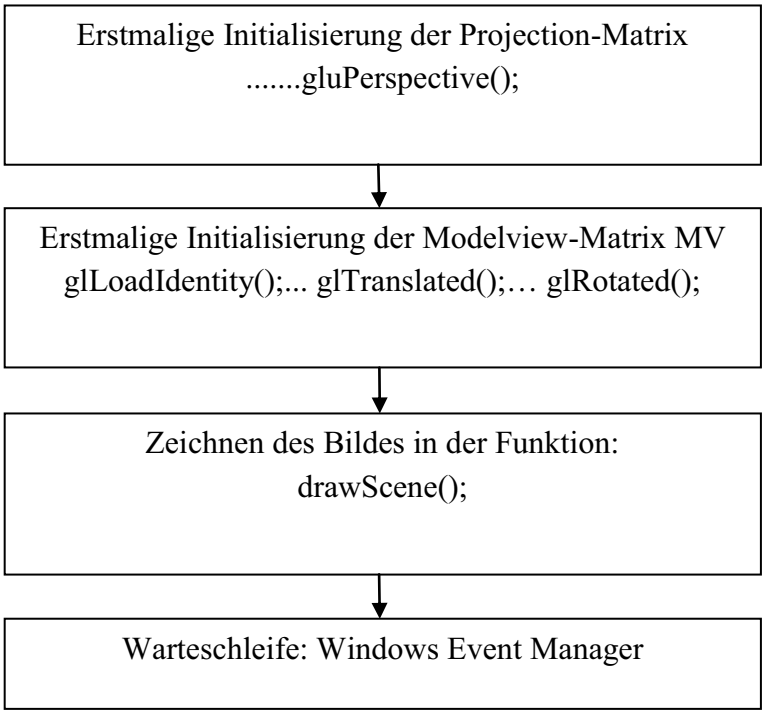

Bild 4: Elementarer Programmablauf

Um die Forderung nach den Mausinteraktionen Panning, Zoom und Rotation zu erfüllen, wird im Folgenden das **Prinzip der Modelview-Matrix-Änderung in einer Maus-Eventfunktion** realisiert. Dieses Prinzip folgt der Literaturstelle /5/.
Im Detail:
Jede Mausbewegung führt zu einer neuen Modelview-Matrix MV_{neu} und mit dieser wird das Bild in drawScene() neu gezeichnet - siehe Bild 5.

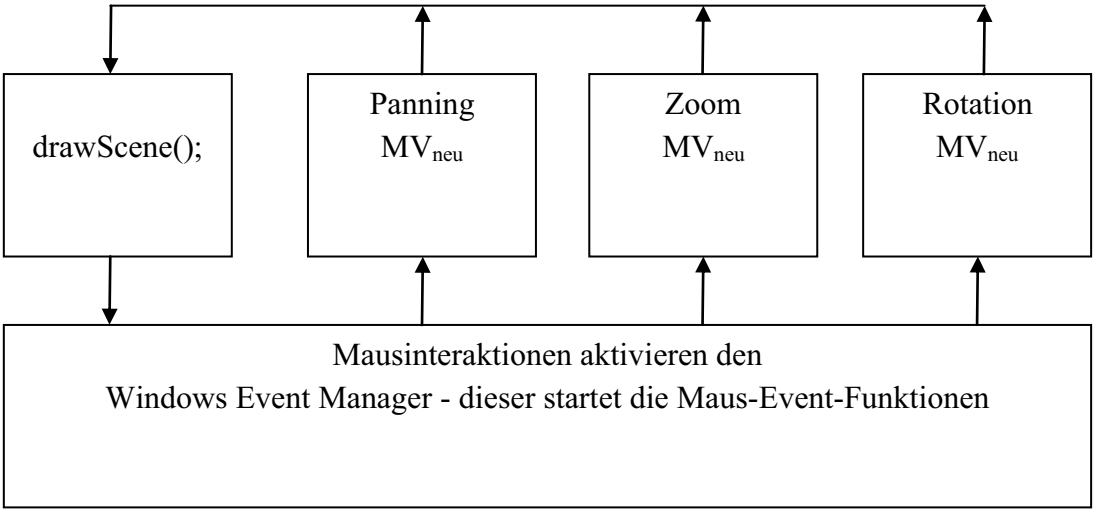

Bild 5: Programmablauf mit Mausinteraktionen

Da die geforderten Mausinteraktionen durch Änderung der Modelview-Matrix erfolgen, ist eine genaue Kenntnis dieser Matrix unerläßlich. Die Modelview-Matrix wird deshalb im Detail im Folgenden angegeben.

5. Mathematik zur Modelview-Matrix MV

Eine sehr gute Darstellung der mathematischen Grundlagen findet man in /1/.

Die **Modelview Matrix** MV hat 16 Elemente.

Eine mögliche Deklaration in C# wäre:
private double[] ModelviewMatrix;
Die zugehörige Initialisierung z.B. in einem Konstruktor:
ModelviewMatrix = new double[16];

Die Feldindizes haben dann folgende Zuordnung:

$$M_v = \begin{bmatrix} 0 & 4 & 8 & 12 \\ 1 & 5 & 9 & 13 \\ 2 & 6 & 10 & 14 \\ 3 & 7 & 11 & 15 \end{bmatrix}$$

Zuerst wird die Modelview Matrix mit der Einheitsmatrix initialisiert:
glLoadIdentity();

$$M_v = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Die drei wichtigen Transformationen in der Computergrafik - Translation, Skalierung und Rotation - werden folgendermaßen in der Modelview-Matrix wirksam:
Die Translation:
glTranslated(x, y, z); ändert die M_v zu:

$$M_v = T = \begin{bmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Die Skalierung:
glScaled(Sx, Sy, Sz);

$$M_v = S = \begin{bmatrix} Sx & 0 & 0 & 0 \\ 0 & Sy & 0 & 0 \\ 0 & 0 & Sz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Die Rotation allgemein: glRotated(α, x, y, z);

$$M_v = R = \begin{bmatrix} R_{11} & R_{12} & R_{13} & 0 \\ R_{21} & R_{22} & R_{23} & 0 \\ R_{31} & R_{32} & R_{33} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Häufig wird um eine konkrete Koordinatenachse gedreht.
Dann lauten die Rotationsmatrizen:

Rotation um α bezüglich der x-Achse:

glRotated(α, 1.0, 0.0, 0.0);

$$M_v = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & +cos\alpha & -sin\alpha & 0 \\ 0 & +sin\alpha & +cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotation um die y-Achse:

glRotated(α, 0.0, 1.0, 0.0);

$$M_v = \begin{bmatrix} cos\alpha & 0 & sin\alpha & 0 \\ 0 & 1 & 0 & 0 \\ -sin\alpha & 0 & cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotation um die z-Achse:

glRotated(α, 0.0, 0.0, 1.0);

$$M_v = \begin{bmatrix} cos\alpha & -sin\alpha & 0 & 0 \\ sin\alpha & cos\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Die Rotationsmatrix für eine beliebige Drehachse (bx, by, bz) im Raum und einen Drehwinkel α lautet (Formel nach /6/ mit Verweis auf /7/):

$$M_v = \begin{bmatrix} c + (1-c) \cdot {b_x}^2 & (1-c) \cdot b_y \cdot b_x - s \cdot b_z & (1-c) \cdot b_z \cdot b_x + s \cdot b_y & 0 \\ (1-c) \cdot b_x \cdot b_y + s \cdot b_z & c + (1-c) \cdot {b_y}^2 & (1-c) \cdot b_z \cdot b_y - s \cdot b_x & 0 \\ (1-c) \cdot b_x \cdot b_z - s \cdot b_y & (1-c) \cdot b_y \cdot b_z + s \cdot b_x & c + (1-c) \cdot {b_z}^2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

mit $c = \cos(\alpha),\ s = \sin(\alpha)$

und wird durch den OpenGL-Befehl

glRotated(α, bx, by, bz);

realisiert.

Bei der Erstellung eines OpenGL-Bildes erfolgen natürlich verschiedene Transformationen nacheinander. Praktisch bedeutet das, dass mehrere Matrizenmultiplikationen aufeinanderfolgen müssen. Im folgenden Beispiel wird gezeigt, wie sich die Modelview-Matrix in einem typischen OpenGL-Programm ändern wird.

Der Ablauf ist:

1. Initialisierung mit der Einheitsmatrix
2. Durchführung einer Rotation
3. Durchführung einer Translation
4. Durchführung einer Skalierung.

Hier die zugehörigen Berechnungen:

1. Initialisierung:

$$M_{v0} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

2. Rotation

$$M_{v1} = M_{v0} \times R = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} R_{11} & R_{12} & R_{13} & 0 \\ R_{21} & R_{22} & R_{23} & 0 \\ R_{31} & R_{32} & R_{33} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} R_{11} & R_{12} & R_{13} & 0 \\ R_{21} & R_{22} & R_{23} & 0 \\ R_{31} & R_{32} & R_{33} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

3. Translation

$$M_{v2} = M_{v1} \times T = \begin{bmatrix} R_{11} & R_{12} & R_{13} & 0 \\ R_{21} & R_{22} & R_{23} & 0 \\ R_{31} & R_{32} & R_{33} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$M_{v2} = \begin{bmatrix} R_{11} & R_{12} & R_{13} & R_{11}x + R_{12}y + R_{13}z \\ R_{21} & R_{22} & R_{23} & R_{21}x + R_{22}y + R_{23}z \\ R_{31} & R_{32} & R_{33} & R_{31}x + R_{32}y + R_{33}z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

4. Skalierung

$$M_{v3} = M_{v2} \times S = \begin{bmatrix} R_{11} & R_{12} & R_{13} & R_{11}x + R_{12}y + R_{13}z \\ R_{21} & R_{22} & R_{23} & R_{21}x + R_{22}y + R_{23}z \\ R_{31} & R_{32} & R_{33} & R_{31}x + R_{32}y + R_{33}z \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} Sx & 0 & 0 & 0 \\ 0 & Sy & 0 & 0 \\ 0 & 0 & Sz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$M_{v3} = \begin{bmatrix} R_{11}Sx & R_{12}Sy & R_{13}Sz & R_{11}x + R_{12}y + R_{13}z \\ R_{21}Sx & R_{22}Sy & R_{23}Sz & R_{21}x + R_{22}y + R_{23}z \\ R_{31}Sx & R_{32}Sy & R_{33}Sz & R_{31}x + R_{32}y + R_{33}z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Im allgemeinen Fall sind (in drawScene()) beliebig viele Rotationen, Skalierungen und Translationen durchgeführt worden. Das Ergebnis aller Matrizenmultiplikationen - die Modelview-Matrix MV - hat aber immer die Form:

$$M_V = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Dabei liefert die linke obere 3*3 Matrix - die Elemente r_{ij} - die zusammengesetzte Rotation und Skalierung, während die Elemente t_x, t_y, t_z die zusammengesetzte Translation angeben. Diese genaue Kenntnis der Modelview-Matrix MV wird - insbesondere bei der Rotation durch Mausinteraktion - noch ausgenutzt.

6. Präzisierung der Aufgabenstellung

Gegeben sei ein OpenGL-Programm, welches zwei Dreiecke nach Bild 6 zeichnet.

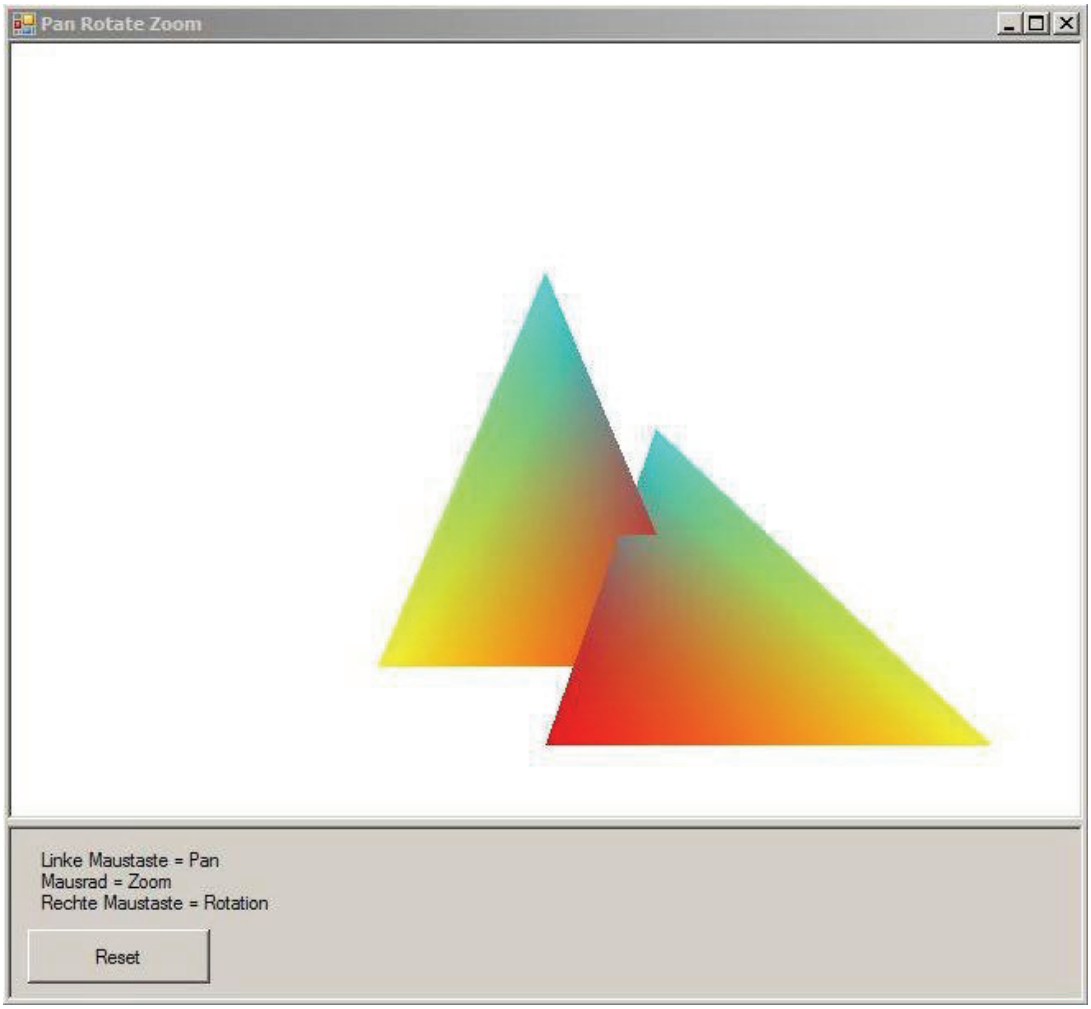

Bild 6: Darstellung zweier Dreiecke

Folgende Anforderungen an die Interaktion mit der Maus werden gestellt:

1. Es soll jederzeit festgestellt werden können, ob sich der Mauszeiger über dem Dreieck oder über dem Hintergrund befindet.
2. Panning: Wird mit der linken Maustaste auf das Dreieck geklickt, soll beim weiteren Ziehen mit der Maus das Dreieck mitbewegt werden.
   Zusatz 1: Die Position des Mauszeigers auf dem Dreieck soll fest bleiben.
   Zusatz 2: Das Panning soll durch die Funktion glTranslated() durchgeführt werden.
3. Zoom: Befindet sich der Mauszeiger über dem Dreieck, soll beim Drehen des Mausrades das Dreieck vergrößert bzw. verkleinert werden.
   Zusatz 1: Der Punkt des Mauszeigers soll lokal fest bleiben, d.h. das Dreiecksstück, über dem sich der Mauszeiger befindet, bleibt immer im Bild.
   Zusatz 2: Das Zoomen soll durch die Funktion glScaled() durchgeführt werden.
4. Rotation: Wird mit der rechten Maustaste auf das Bild geklickt, soll sich beim weiteren Ziehen mit der Maus das Dreieck drehen.

Zusatz 1: Horizontale Mausbewegung dreht um die senkrechte Achse, vertikale Mausbewegung dreht um die waagerechte Achse.
Zusatz 2: Die Rotation soll durch die Funktion glRotated() durchgeführt werden.

7. Grundideen

Grundidee 1:
Beim Betätigen einer Maustaste befindet sich der Cursor an einer bestimmten Position auf dem Bildschirm - dies sind 2D-GDI-Pixelkoordinaten - oder auch die Mauskoordinaten. Zur Verarbeitung in OpenGL werden diese 2D-GDI-Mauskoordinaten in Weltkoordinaten der Szene umgerechnet.

Grundidee 2:
Die aktuelle Modelview-Matrix wird in einer Maus-Eventfunktion geändert und dann drawScene() mit der geänderten Modelview-Matrix aufgerufen.

Anforderung 1: Ermittlung der Weltkoordinaten am Mausklickpunkt

Maustastendrücke und die Mausbewegungen werden in jedem Windows-Programm durch Eventfunktionen ausgewertet. Beim Eintritt in eine Mauseventfunktion stellt das Betriebssystem die aktuellen 2D-Koordinaten der Maus als zwei int-Zahlen, gemessen in Bildschirmpixeln und bezogen auf die linke obere Ecke des aktuellen Fensters (GDI-Koordinaten), bereit.
Diese Pixelkoordinaten sind nun auf Weltkoordinaten umzurechnen. Dazu stellt OpenGL die Funktion

```
int gluUnProject (
GLdouble winx, GLdouble winy, GLdouble winz, // die Mauskoordinaten
const GLdouble modelMatrix[16],
const GLdouble projMatrix[16],
const GLint viewport[4],
GLdouble *objx, GLdouble *objy, GLdouble *objz // die Weltkoordinaten
); bereit.
```

Eingangsdaten sind prinzipiell immer:
-die Modelview-Matrix MV
-die Projection-Matrix und der
-aktuelle Viewport

Zusätzlich müssen die 2D-GDI-Mauspixelkoordinaten – allerdings als 3D-OpenGL-Windowskoordinaten übergeben werden. Dabei fehlt natürlich eine z-Koordinate - der z-Buffer Wert - die Mauskoordinaten liegen ja nur in 2D vor.
Zu unterscheiden ist also die Umrechnung der x, y-GDI-Mauspixelkoordinaten von der Berechnung der z-Windowskoordinate – diese ist für die Funktion gluUnProject() der

Tiefenwert aus dem Tiefenpuffer, welcher, wie bereits ausgeführt, float-Daten, begrenzt auf den Bereich 0.0 bis 1.0 enthält.

Es gilt für den z-Tiefenwert:
0.0 = near-Ebene
1.0 = far-Ebene Hinweis: der Bereich ist den Weltkoordinaten nicht linear zugeordnet!

Als Ausgangsdaten liefert die Funktion gluUnProject() die 3D-Weltkoordinaten in objx, objy, objz zurück. Folgende Arbeitsschritte sind also erforderlich:

Schritt 1:
Umrechnung der x, y-GDI-Mauspixelkoordinaten in x, y -OpenGL-Windowskoordinaten

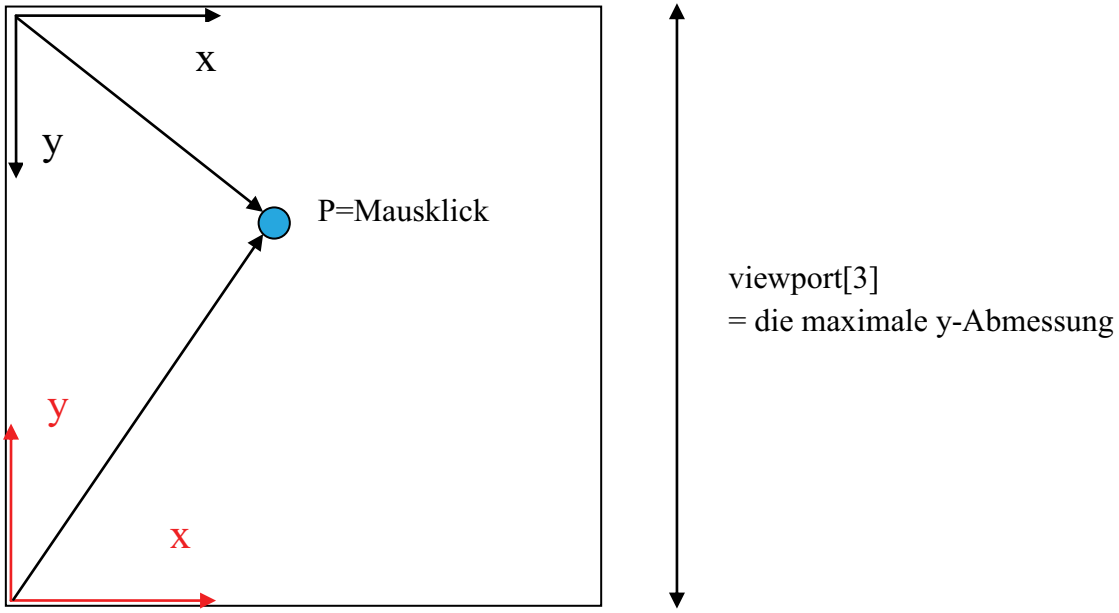

Bild 7: Maus-GDI- und Windowskoordinaten

Im Bild 7 liefert eine Windows-Mauseventfunktion die "schwarzen" int-x,y-GDI-Koordinaten für den Mausklick am Punkt P. Die "roten" int-OpenGL Windowskoordinaten lauten dann:

$x_{rot} = x_{schwarz}$
$y_{rot} = viewport[3] - y_{schwarz} - 1$

Schritt 2:
Ermittlung des z-Buffer-Wertes (die OpenGL-z-Koordinate) am Mausklickpunkt

Die Ermittlung des z-Buffer-Wertes erfolgt durch die Nutzung der OpenGL-Funktion **glReadPixels().**
Diese kann für 1 Bildschirmpixel den dazugehörigen normierten z-Buffer-Wert liefern.

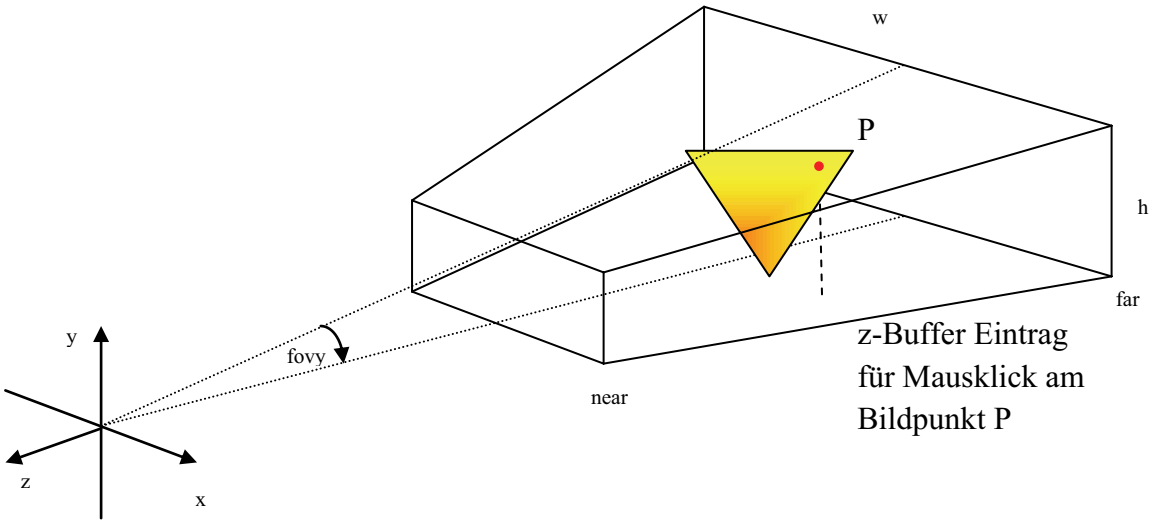

Bild 8: Der z-Buffer Wert am Bildpunkt P

Um den z-Buffer Eintrag (der Wert tiefe) für den Punkt P an der Mausposition im Bild 8 zu ermitteln
geht man wie folgt vor:

float tiefe;
glReadPixels(x_{rot}, y_{rot}, 1, 1, GL_DEPTH_COMPONENT, GL_FLOAT, &tiefe);

Auch hier sind also die "roten" OpenGL-Windowskoordinaten zu verwenden.
Hinweis: Liegen mehrere Objekte hintereinander, wird nur der Wert für das vorderste Objekt geliefert. Dies sollte aber meist ja auch beabsichtigt sein.

Zusammenfassung zur Ermittlung der Weltkoordinaten am Mausklickpunkt

Folgende Funktion liefert in einem OpenGL-Programm für 2D-Windows-Mauseventkoordinaten die zugehörigen 3D-OpenGL-Weltkoordinaten:

```
private bool MouseToWorld(int Mouse_X, int Mouse_Y, ref double
WorldMouse_X, ref double WorldMouse_Y, ref double WorldMouse_Z)
        {
            int[] viewport = new int[4];
            double[] ModelviewMatrix = new double[16];
            double[] ProjectionMatrix = new double[16];
            int OGL_Y;
            // Matrizen auslesen
            Gl.glGetIntegerv(Gl.GL_VIEWPORT, viewport);
            Gl.glGetDoublev(Gl.GL_MODELVIEW_MATRIX, ModelviewMatrix);
            Gl.glGetDoublev(Gl.GL_PROJECTION_MATRIX, ProjectionMatrix);
            // Richtungsumkehr der y-Achse
            OGL_Y = viewport[3] - Mouse_Y - 1;
            float[] tiefe = new float[1];
            tiefe[0] = 0.0f;
```

```
            Gl.glReadPixels(Mouse_X, OGL_Y, 1, 1, Gl.GL_DEPTH_COMPONENT,
            Gl.GL_FLOAT, tiefe);
            Glu.gluUnProject((double)Mouse_X, (double)OGL_Y,
            (double)tiefe[0], ModelviewMatrix, ProjectionMatrix, viewport,
            out WorldMouse_X, out WorldMouse_Y, out WorldMouse_Z);
            if (tiefe[0] > 0.9999)
                return false;  // nicht auf Objekt geklickt
            else
                return true;
        }
```

Anforderung 2: Panning

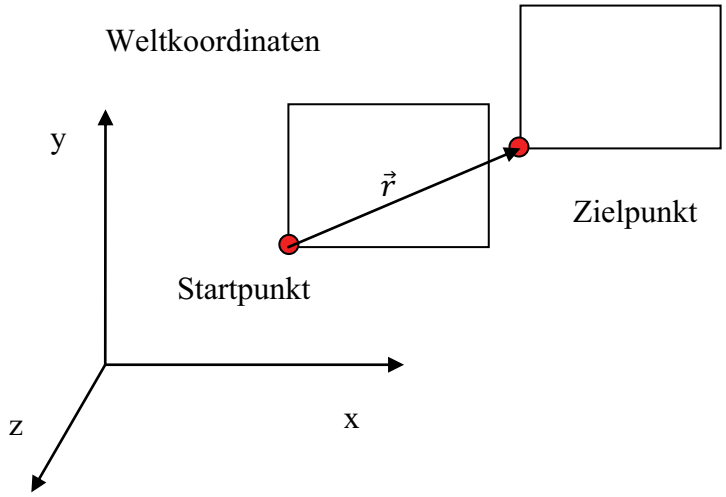

Bild 9: Prinzip des Panning

Aufgabe: Das Objekt Rechteck im Bild 9 soll um den Vektor $\vec{r}$ verschoben werden.

Lösung: Wenn die Koordinaten des Vektors $\vec{r}$ als Weltkoordinaten rx, ry, rz bekannt sind, kann das Panning einfach als Translations-Transformation programmiert werden. Da die Mauskoordinaten ja nur 2D-Koordinaten sind, wird die z-Koordinate auf Null gesetzt.

glTranslated(rx, ry, 0.0);

Praktische Lösung:
Die Weltkoordinaten des Startpunktes werden aus den Mauskoordinaten beim Drücken der linken Maustaste ausgerechnet und in den Memberkoordinaten weltMaus_x_Start, weltMaus_y_Start , weltMaus_z_Start gespeichert.

```
private void TaoOGLControl_MouseDown(object sender, MouseEventArgs e)
{
   xStart = e.X;
   yStart = e.Y;
   // Berechnung der Weltkoordinaten am Mausklick-Punkt
   // Wird kein Objekt getroffen ( also wird der Hintergrund-die far-Ebene-
   // getroffen ) sind die Weltkoordinaten Null
```

```
   if (MouseToWorld(xStart, yStart, ref weltMaus_x_Start, ref
       weltMaus_y_Start, ref weltMaus_z_Start) == false)
      {
       weltMaus_x_Start = 0.0;
       weltMaus_y_Start = 0.0;
       weltMaus_z_Start = 0.0;
      }
}
```

Beim Bewegen der Maus – also dem Panning – werden die Weltkoordinaten des Zielpunktes weltMaus_x, welt Maus_y, welt Maus_z berechnet, die Differenz zum Startpunkt gebildet, die Translations-Transformation ausgeführt und das Bild neu gezeichnet. Ausgenutzt wird die Windows-Eventfunktion MouseMove() (hier wird nur der Panning-Teil angegeben, der Rotations-Teil folgt später).

```
private void TaoOGLControl_MouseMove(object sender, MouseEventArgs e)
{
 // Pan
 if (e.Button == MouseButtons.Left)
  {
   double weltMaus_x = 0.0;
   double weltMaus_y = 0.0;
   double weltMaus_z = 0.0;
   // Pan nur, wenn ein Objekt und nicht die far-Ebene getroffen wird
   if (MouseToWorld(e.X, e.Y, ref weltMaus_x, ref weltMaus_y,
       ref weltMaus_z) == true)
    { // hier erfolgt das Panning
     Gl.glTranslated(weltMaus_x - weltMaus_x_Start,
                     weltMaus_y - weltMaus_y_Start, 0.0);
     this.SimpleOpenGlControl.Refresh(); // Bild neu zeichnen
    }
  }
}
```

Anforderung 3: Zoom

Das Zoomen des Bildes wird durch die Transformation Skalierung durchgeführt. Die besondere Forderung dabei ist, dass die mit der Maus angeklickte Objekt-Position immer im Bild bleiben soll, der Rest des Objektes kann dann beim Zoomen (Vergrößern) aus dem Bild herauswandern.

Lösung:

- Translation des Koordinatensystems zum Mausklickpunkt im Objekt
- Skalierung (Zoom)
- Translation des skalierten Koordinatensystems wieder zurück zum Ausgangspunkt

konkret:

Die Funktion Zoom wird in der Eventfunktion MouseWheel() realisiert.

```
private void TaoOGLControl_MouseWheel(object sender, MouseEventArgs e)
{
 // Die momentane Mausposition
 xStart = e.X;
 yStart = e.Y;
 // lokale Variablen für die zugehörigen Weltkoordinaten
 double weltMaus_x = 0.0;
 double weltMaus_y = 0.0;
```

```
 double weltMaus_z = 0.0;
 // Berechnung dieser zugehörigen Weltkoordinaten
 // true, falls ein Objekt getroffen wird.
 if (MouseToWorld(xStart, yStart, ref weltMaus_x, ref weltMaus_y,
                  ref weltMaus_z) == true)
 {
  // eine e-Funktion zum intuitiven Zoomen
  // wenn e.Delta = -120 dann f = 0.9 d.h. Verkleinern = Zoom Out
  // wenn e.Delta = +120 dann f = 1.1 d.h. Vergrößern = Zoom In
  double f = Math.Exp(0.1 * (double)e.Delta / 120.0);
  // Gesamtes Zoomen speichern, um es in drawScene() rückgängig machen
  // zu können
  absoluterZoomfaktor = absoluterZoomfaktor * f;
  // Koordinatensystem zum Mauspunkt verschieben
  // damit bleibt der getroffene Objektpunkt beim Zoomen im Bild an
  // der gleichen Stelle
  Gl.glTranslated(weltMaus_x, weltMaus_y, weltMaus_z);
  // Hier wird gezoomt!!
  Gl.glScaled(f, f, f);
  // Translation des Koordinatensystems rückgängig machen
  Gl.glTranslated(-weltMaus_x, -weltMaus_y, -weltMaus_z);
  // Bild neu zeichnen
  this.SimpleOpenGlControl.Refresh();
 }
}
```

Anforderung 4: Rotation

Aufgabenstellung:

Der Anwender sieht das Objekt - hier einen gelben Quader im Bild 10 - auf dem Bildschirm und möchte es z.B. um eine bezüglich des Bildschirms senkrechte Achse - hier angegeben durch den Vektor $\vec{a}$ - drehen. Der Drehwinkel soll den Wert angle haben. Dazu bewegt er intuitiv die Maus waagerecht - hier vom Punkt P_{m1} zum Punkt P_{m2}.
Grundprinzip nach /5/.

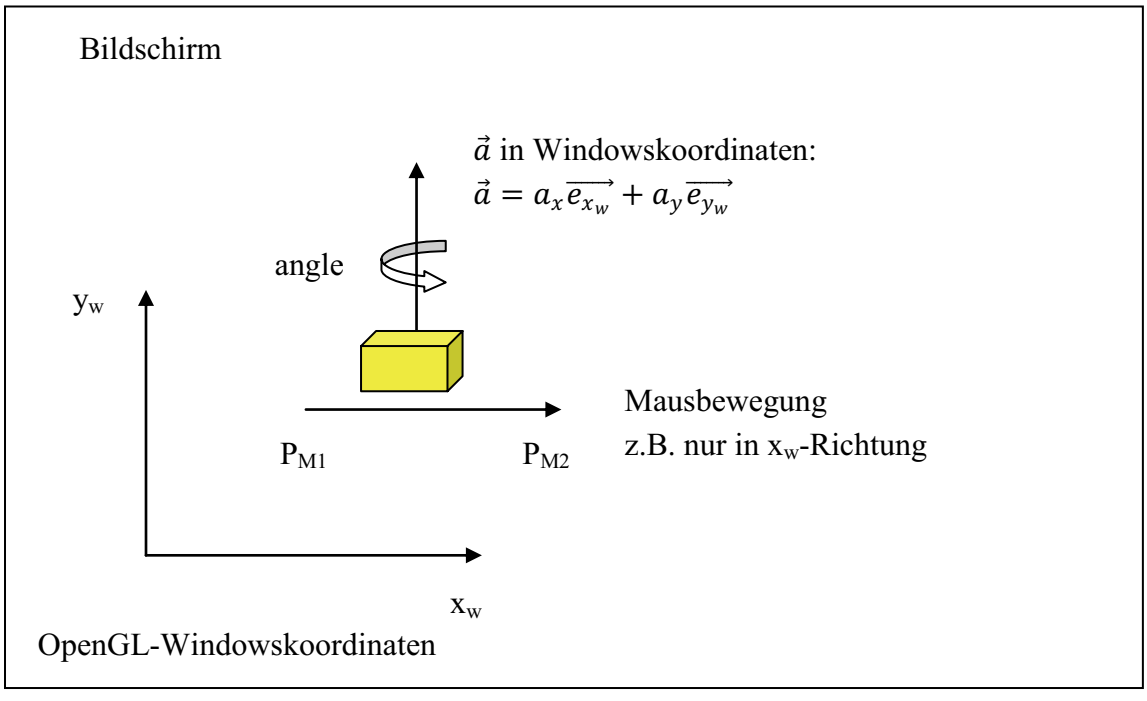

Bild 10: Rotation nach Mausbewegung

Für eine beliebige Drehung eines Objektes mit der Maus formuliert man nach Bild 11:

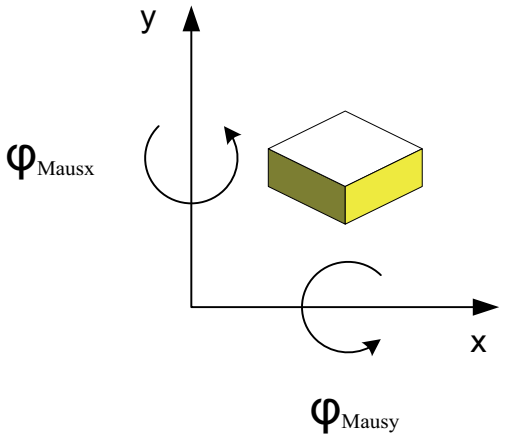

Bild 11: Rotation um die x- und y-Achse

φ_{Mausx}: Bei einer Mausbewegung in positiver x-Richtung auf dem Bildschirm (waagerechte Bewegung) soll eine Drehung um die y-Achse (der Windowskoordinaten y_w) in mathematisch positiver Drehrichtung erfolgen. Im Vektor $\vec{a}$ wird deshalb die Mausbewegung in x-Richtung zur Berechnung der Koordinate ay benutzt (entsprechend für ax).

φ_{Mausy}: Bei einer Mausbewegung in negativer y-Richtung der Windowskoordinaten y_w (d.h. auf dem Bildschirm vertikale Bewegung der Maus nach unten) soll eine Drehung um die x-Achse (der Windowskoordinaten x_w) in mathematisch positiver Drehrichtung erfolgen.

Die Koordinaten des Vektors $\vec{a}$ werden aus den Mauskoordinaten wie folgt berechnet (beachte den Unterschied zwischen Windows- und Mauskoordinaten!):
P_{M1}, P_{M2} – Start- und Endpunkt der Mausbewegung in Pixeln

$$ax = P_{M2_y} - P_{M1_y}$$

$$ay = P_{M2_x} - P_{M1_x}$$

Der Drehwinkel angle (in Grad) wird nach folgender "Hilfsformel" berechnet (angle könnte zum Testen auch einfach eine Konstante z.B. 1 Grad, sein):
Mit b – Fensterbreite in Pixeln:

$$angle = \frac{\sqrt{ax^2 + ay^2}}{b} \cdot 180$$

In OpenGL wird das Objekt in Weltkoordinaten des (bereits beliebig transformierten) Koordinatensystems x', y', z' beschrieben.
Um die Drehung des Objektes in OpenGL zu programmieren, benötigt man die Umrechnung der Windowskoordinaten des Vektors $\vec{a}$ in seine Weltkoordinaten.

Annahme: Die Weltkoordinaten des Vektors $\vec{a}$ sind im Vektor $\vec{b}$ gespeichert. Kennt man den Vektor $\vec{b}$ kann man die Drehung leicht als Rotation aufschreiben:

glRotated(angle, bx, by, bz);

Damit besteht die Hauptaufgabe jetzt in der Berechnung der Koordinaten bx, by, bz, also in der Berechnung des Vektors $\vec{b}$.
Hinweis: Im nachfolgenden Bild 12 liegt der Vektor $\vec{b}$ der Einfachheit halber in der y' - z' - Ebene.

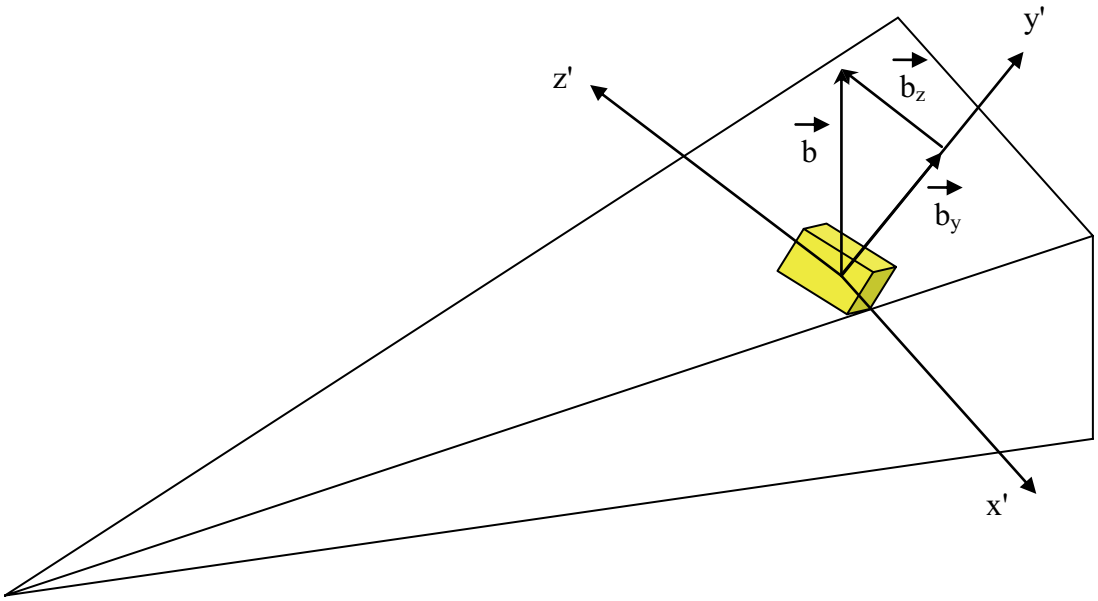

Bild 12: Rotation um eine beliebige Achse

Der Weg dazu ist relativ leicht, denn man weiß ja, wie in OpenGL Windowskoordinaten aus Weltkoordinaten berechnet werden. Dies erfolgt durch die Multiplikation der Weltkoordinaten mit der Modelview-Matrix. Zur Erinnerung, die Modelview-Matrix enthält den momentanen Zustand aller durchgeführten Bildtransformationen (Translation, Rotation, Skalierung). Um aus den Windowskoordinaten nun Weltkoordinaten zu berechnen, die Rechnung also umzukehren, multipliziert man die Windowskoordinaten mit der inversen Modelview-Matrix.

$$\vec{b} = Mv^{-1} \cdot \vec{a}$$

Zu berechnen ist also die inverse Modelview-Matrix.

Ist die inverse Modelview-Matrix berechnet, so lautet obige Formel konkret:

$$\begin{bmatrix} b_x \\ b_y \\ b_z \\ 0 \end{bmatrix} = \begin{bmatrix} {r_{11}}^{-1} & {r_{12}}^{-1} & {r_{13}}^{-1} & {t_x}^{*-1} \\ {r_{21}}^{-1} & {r_{22}}^{-1} & {r_{23}}^{-1} & {t_y}^{*-1} \\ {r_{31}}^{-1} & {r_{32}}^{-1} & {r_{33}}^{-1} & {t_z}^{*-1} \\ 0 & 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} a_x \\ a_y \\ 0 \\ 0 \end{bmatrix}$$

Unter Berücksichtigung der rechnerinternen Matrix-Indizierung:

ModelviewMatrixInvers[0]=r_{11}^{-1}, ModelviewMatrixInvers[4]=r_{12}^{-1}
ModelviewMatrixInvers[1]=r_{21}^{-1}, ModelviewMatrixInvers[5]=r_{22}^{-1}
ModelviewMatrixInvers[2]=r_{31}^{-1}, ModelviewMatrixInvers[6]=r_{32}^{-1}

ist also zu berechnen:

bx = ModelviewMatrixInvers[0] · ax + ModelviewMatrixInvers[4] · ay;
by = ModelviewMatrixInvers[1] · ax + ModelviewMatrixInvers[5] · ay;
bz = ModelviewMatrixInvers[2] · ax + ModelviewMatrixInvers[6] · ay;

Damit sich das Objekt (hier der gelbe Quader) um seinen Mittelpunkt dreht - was sicher intuitiv erwartet wird - muss man vor der Rotation das aktuelle Weltkoordinatensystem in den Mittelpunkt des Objektes verschieben. Wenn die x-y-Mittelpunktskoordinaten des Objektes xCenter und yCenter lauten, ist also zu programmieren:

glTranslated(xCenter, yCenter, 0.0);
glRotated(angle, bx, by, bz);
glTranslated(-xCenter, -yCenter, 0.0);

Diese drei Funktionsaufrufe werden in der Mouse-Event-Funktion MouseMove() durchgeführt.

```
private void TaoOGLControl_MouseMove(object sender, MouseEventArgs e)
{
 // Hinweis: Pan ebenfalls in MouseMove
 // Rotate mit rechter Maustaste
 if (e.Button == MouseButtons.Right)
 {
  // 1.Drehwinkel angle in Grad aus der Mausbewegung bestimmen
  double ax, ay, az;
  double bx, by, bz;
  double angle;
  ax = e.Y - yStart;
  ay = e.X - xStart;
  az = 0.0;
  int[] viewport = new int[4];
  Gl.glGetIntegerv(Gl.GL_VIEWPORT, viewport);
  angle = Math.Sqrt(ax * ax + ay * ay + az * az) / (double)(viewport[2]
  + 1) * 180.0;
  double [] ModelviewMatrix_Speicher;
```

```
  ModelviewMatrix_Speicher = new double[16];
  Gl.glGetDoublev(Gl.GL_MODELVIEW_MATRIX, ModelviewMatrix_Speicher);
  double[] ModelviewMatrixInvers;
  ModelviewMatrixInvers = new double[16];
  __gluInvertMatrixd(ModelviewMatrix_Speicher, ModelviewMatrixInvers);
  bx = ModelviewMatrixInvers[0] * ax + ModelviewMatrixInvers[4] * ay
  + ModelviewMatrixInvers[8] * az;
  by = ModelviewMatrixInvers[1] * ax + ModelviewMatrixInvers[5] * ay
  + ModelviewMatrixInvers[9] * az;
  bz = ModelviewMatrixInvers[2] * ax + ModelviewMatrixInvers[6] * ay
  + ModelviewMatrixInvers[10] * az;
  // Hier erfolgt endlich die Rotation
  Gl.glTranslated(xCenter, yCenter, 0.0);
  Gl.glRotated(angle, bx, by, bz);
  Gl.glTranslated(-xCenter, -yCenter, 0.0);
  xStart = e.X;
  yStart = e.Y;
  this.SimpleOpenGlControl.Refresh();
}
```

Der Funktionsaufruf

```
__gluInvertMatrixd(ModelviewMatrix_Speicher, ModelviewMatrixInvers);
```

realisiert die Berechnung der erforderlichen Elemente der inversen Modelview-Matrix und wurde aus /3/ übernommen.

8. Zusammenfassung

Der Artikel beschreibt die Realisierung der Mausinteraktionen Panning, Zoom und Rotation in einem Windows-Programm mit einem OpenGL-Fenster. Es wird deutlich, dass eine genaue Kenntnis der zugrundeliegenden Computergrafik - Mathematik erforderlich ist. Deshalb werden insbesondere die notwendigen Matrizentransformationen der Modelview-Matrix exakt angegeben. Der Quelltext des Programmes befindet sich in der Anlage.

9. Literaturverzeichnis

/1/ Foley, James D.; van Dam, Andries; Feiner, Steven K.; Hughes, John F.; Phillips, Richard L.
Grundlagen der Computergraphik
Addison-Wesley, 1994
ISBN 3-89319-647-1

/2/ Shreiner, Dave; Woo, Mason; Neider, Jackie; Davis, Tom
OpenGL Programming Guide Sixth Edition The Official Guide to Learning OpenGL, Version 2.1
Addison-Wesley, 2007
ISBN 0-321-48100-3

/3/ http://www.mesa3D.org
Mesa 3D Graphics Library: Mesa 7.1.5
enthält Funktionen von SGI: oss.sgi.com/projects/FreeB

/4/ Tao Framework 2.1.0 Released
http://sourceforge.net/projects/taoframework

/5/ GLT ZPR - Zoom, Pan and Rotate
www.nigels.com/glt/gltzpr

/6/ Gruber, Diana
The Mathematics of the 3D Rotation Matrix
http://www.fastgraph.com/makegames/3drotation

/7/ Glassner, Andrew, S.
Graphics Gems
Academic Press, Inc. Orlando, FL, USA 1990
ISBN:0122861655

10. Kontakt:

Marion.Braunschweig@TU-Ilmenau.de
Mathias.Weiss@TU-Ilmenau.de

11. Anlage Programmquelltext

```
// Visual Studio 2008, C#, Tao, Windows Application
// Das Tao Control: "SimpleOpenGlControl" wird benutzt
// http://sourceforge.net/projects/taoframework

using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

// nötig für Tao ( Referenzen sind zu addieren )
using Tao.OpenGl;
using Tao.Platform.Windows;


namespace PanRotateZoomVeroeffentlichung
{
    public partial class TaoOpenGLPresentationLayer : Form
    {
        /// <summary>
        /// Beim Drücken einer Maustaste werden die momentanen 2D-Mauskoordinaten
        /// (Bildschirmpixel = GDI-Windowskoordinaten, bezogen auf die linke obere
        /// Windowsecke)
        /// in xStart und yStart gespeichert. Wird die Maus dann bei gedrückter Taste bewegt,
        /// kann die Wegstrecke als Differenz zwischen neuen Mauskoordinaten und den
        /// Startpunktskoordinaten berechnet werden.
        /// </summary>
        private int xStart = 0;
        private int yStart = 0;
        /// <summary>
        /// Die 2D-Mauskoordinaten xStart, yStart werden in 3D-OpenGL-Weltkoordinaten
        /// des Darstellungsraumes umgerechnet und in den folgenden Koordinaten gespeichert.
        /// </summary>
        private double weltMaus_x_Start = 0.0;
        private double weltMaus_y_Start = 0.0;
        private double weltMaus_z_Start = 0.0;
       /// <summary>
       /// Um jederzeit alles Zoomen rückgängig machen zu können, werden alle Zoomoperationen
       /// in der Variablen absoluterZoomfaktor gespeichert
       /// </summary>
        private double absoluterZoomfaktor = 1.0;
        /// <summary>
        /// Die Koordinaten des Dreiecksmittelpunktes - hier Null
        /// </summary>
        private double xCenter = 0.0;
        private double yCenter = 0.0;


        public TaoOpenGLPresentationLayer()
        {
            // Default: Vom Wizard eingefügt
            InitializeComponent();
            // Manuell eingefügt: Tao Initialisierungen
            this.SimpleOpenGlControl.InitializeContexts();
            // Manuell eingefügt: Initialize()
            OpenGLInitialize();
            Panel2Text.Text = "Linke Maustaste = Pan\nMausrad = Zoom\nRechte Maustaste =
            Rotation";
        }

        /// <summary>
        /// Einmalige Initialisierungen
        /// </summary>
        private void OpenGLInitialize()
        {
            Gl.glClearColor(1.0f, 1.0f, 1.0f, 1.0f);
            Gl.glEnable(Gl.GL_DEPTH_TEST);
            InitProjectionMatrix();
            InitModelviewMatrix();
```

```
}

private void InitProjectionMatrix()
{
    double fovy = 45.0;   // Öffnungswinkel
    int hoehe = this.SimpleOpenGlControl.Height;
    int breite = this.SimpleOpenGlControl.Width;
    double aspect = 0.0;     //Seitenverhaeltnis des Fensters
    if (hoehe == 0)
        hoehe = 1;
    //Das ganze Fenster soll der Viewport sein.
    Gl.glViewport(0, 0, breite, hoehe);
    Gl.glMatrixMode(Gl.GL_PROJECTION);
    Gl.glLoadIdentity();
    aspect = (double)breite / (double)hoehe;
    // Demo Werte: near = 3 m far = 10 m
    Glu.gluPerspective(fovy, aspect, 3.0, 10.0);
    // MV-Matrix sei wieder die Current Matrix
    Gl.glMatrixMode(Gl.GL_MODELVIEW);
}

private void InitModelviewMatrix()
{
    Gl.glMatrixMode(Gl.GL_MODELVIEW);
    Gl.glLoadIdentity();
    // Erste Transformation:
    //vor DrawScene() - in Sichtbarkeitsbereich ( 3m bis 10 m ) gehen und etwas drehen
    Gl.glTranslated(0.0, 0.0, -7.0);
    //Kippen um die x-Achse
    Gl.glRotated(-45.0, 1.0, 0.0, 0.0);
    // die MV-Matrix bleibt Current Matrix
}

/// <summary>
/// drawScene() neu: die MV-Matrix ist für den Darstellungsraum gesetzt
/// Event Wizard: Appearance - Paint
/// gezeichnet wird ein Dreieck
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void TaoOGLControl_Paint(object sender, PaintEventArgs e)
{
    Gl.glPushMatrix();
    Gl.glClear(Gl.GL_COLOR_BUFFER_BIT | Gl.GL_DEPTH_BUFFER_BIT);
    // Ein Dreieck
    Gl.glBegin(Gl.GL_TRIANGLES);
    Gl.glColor3d(1.0, 1.0, 0.0); //Gelb
    Gl.glVertex3d(-1.0, -2.0, 0.0);
    Gl.glColor3d(1.0, 0.0, 0.0); //Rot
    Gl.glVertex3d(1.0, -2.0, 0.0);
    Gl.glColor3d(0.0, 1.0, 1.0); //Cyan
    Gl.glVertex3d(0.0, 2.0, 0.0);
    Gl.glEnd();

    Gl.glTranslated(1.0, -1.0, 0.0);

    // Ein zweites Dreieck
    Gl.glBegin(Gl.GL_TRIANGLES);
    Gl.glColor3d(1.0, .0, 0.0); //Rot
    Gl.glVertex3d(-1.0, -2.0, 1.0);
    Gl.glColor3d(1.0, 1.0, 0.0); //Gelb
    Gl.glVertex3d(1.0, -2.0, 1.0);
    Gl.glColor3d(0.0, 1.0, 1.0); //Cyan
    Gl.glVertex3d(0.0, 2.0, -1.0);
    Gl.glEnd();

    Gl.glPopMatrix();
}

/// <summary>
/// Aufruf nach Fenstergrößenänderungen
/// Event Wizard: Layout - Resize
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void TaoOGLControl_Resize(object sender, EventArgs e)
{
    InitProjectionMatrix();
```

```
}

// Ende Tao-Gerüst

////////////////////////////////////////////////////////////////////////////
// Interaktionen mit der Maus
// Die Mouse-Events werden vom SimpleOpenGlControl abgefangen - nicht von Form!
////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////
// Zoom mit MouseWheel
////////////////////////////////////////////////////////////////////////////
private void TaoOGLControl_MouseWheel(object sender, MouseEventArgs e)
{
    // Die momentane Mausposition
    xStart = e.X;
    yStart = e.Y;

     double weltMaus_x = 0.0;
     double weltMaus_y = 0.0;
     double weltMaus_z = 0.0;

    // Ermittlung der zugehörigen Weltkoordinaten
    // true, falls ein Objekt getroffen wird.
    if (MouseToWorld(xStart, yStart, ref weltMaus_x, ref weltMaus_y, ref weltMaus_z)
        == true)
    {
        // Wie Google und Adobe
        // wenn e.Delta = -120 dann f = 0.9 d.h. Verkleinern = Zoom Out,
        // WheelRad wird mathematisch negativ gedreht
        // wenn e.Delta = +120 dann f = 1.1 d.h. Vergrößern = Zoom In

        double f = Math.Exp(0.1 * (double)e.Delta / 120.0);
        // Gesamtes Zoomen speichern, um es in drawScene() rückgängig machen zu können
        absoluterZoomfaktor = absoluterZoomfaktor * f;

        // Koordinatensystem zum Mauspunkt verschieben
        // damit bleibt der getroffene Objektpunkt beim Zoomen im Bild
        // an der gleichen Stelle
        Gl.glTranslated(weltMaus_x, weltMaus_y, weltMaus_z);
        // Hier wird gezoomt!!
        Gl.glScaled(f, f, f);
        // Translation des Koordinatensystems rückgängig machen
        Gl.glTranslated(-weltMaus_x, -weltMaus_y, -weltMaus_z);
        // Bild neu zeichnen
        this.SimpleOpenGlControl.Refresh();
    }
}

////////////////////////////////////////////////////////////////////////////
// Pan = Ziehen mit linker Taste: benötigt MouseDown und MouseMove
// Rotate = Ziehen mit rechter Maustaste
////////////////////////////////////////////////////////////////////////////


/// <summary>
/// Irgendeine Maustaste wird gedrückt
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void TaoOGLControl_MouseDown(object sender, MouseEventArgs e)
{


        xStart = e.X;
        yStart = e.Y;
        // Berechnung der Weltkoordinaten am Mausklick-Punkt
        // Wird kein Objekt getroffen ( also wird der Hintergrund-die far-Ebene-
        // getroffen ) sind die Koordinaten Null
        if (MouseToWorld(xStart, yStart, ref weltMaus_x_Start, ref weltMaus_y_Start,
             ref weltMaus_z_Start) == false)
        {
            weltMaus_x_Start = 0.0;
            weltMaus_y_Start = 0.0;
            weltMaus_z_Start = 0.0;
        }

}
```

```
/// <summary>
/// Ziehen der Maus wird für Pan und Rotate ausgenutzt
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void TaoOGLControl_MouseMove(object sender, MouseEventArgs e)
{
    // Pan
    if (e.Button == MouseButtons.Left)
    {
         double weltMaus_x = 0.0;
         double weltMaus_y = 0.0;
         double weltMaus_z = 0.0;

        // Pan nur, wenn ein Objekt und nicht die far-Ebene getroffen wird
        if (MouseToWorld(e.X, e.Y, ref weltMaus_x, ref weltMaus_y, ref weltMaus_z)
            == true)
        {
            Gl.glTranslated(weltMaus_x - weltMaus_x_Start, weltMaus_y -
            weltMaus_y_Start, 0.0);
            this.SimpleOpenGlControl.Refresh();
        }
    }

    // Rotate
    if (e.Button == MouseButtons.Right)
    {
        // 1.Drehwinkel angle in Grad aus der Mausbewegung bestimmen
        double ax, ay, az;
        double bx, by, bz;
        double angle;
        ax = e.Y - yStart;
        ay = e.X - xStart;
        az = 0.0;
        // Der Viewport wird gesetzt mit
        // glViewport( int x, int y, int width, int height); siehe oben.
        // x,y = Koordinaten des Ursprungs bezogen auf die linke untere Ecke
        // des Windows
        // width, heigt = Breite und Höhe des Windows
        // alle Angaben in Pixeln
        // x zeigt nach links
        // y zeigt nach oben
        // die Koordinaten eines Pixels im Window heißen "window coordinates"
        // in Pixeln
        int[] viewport = new int[4];
        /* viewport[0] = x - Ursprung
         * viewport[1] = y - Ursprung
         * viewport[2] = width  - die x-Richtung
         * viewport[3] = height - die y-Richtung
         */
        Gl.glGetIntegerv(Gl.GL_VIEWPORT, viewport);
        // Eine Zahl angle, die als Winkel definiert wird
        // Parameter: die Mausbewegung und die Fensterbreite aus viewport[2]
        // Zahlenbeispiele:
        // Minimum ax = 1 Pixel, viewport[2] = 1680 Pixel -> angle = 0,1 Grad
        // sehr schnelle Mausbewegungen führen zu angle ca. 10 bis 12 Grad
        // Vorteil der Formel: die Mausgeschwindigkeit und die Bildgröße
        // werden berücksichtigt
        // die Addition von 1 Pixel ( viewport[2] + 1 ) verhindert die
        // Division durch Null
        angle = Math.Sqrt(ax * ax + ay * ay + az * az) /
                        (double)(viewport[2] + 1) * 180.0;

        // 2. Drehachse bestimmen
        // Abfrage der aktuellen Modelviematrix
        double [] ModelviewMatrix_Speicher;
        ModelviewMatrix_Speicher = new double[16];
        Gl.glGetDoublev(Gl.GL_MODELVIEW_MATRIX, ModelviewMatrix_Speicher);

        // Berechnung der inversen Modelviewmatrix
        // Mathematik aus der Literatur
        double[] ModelviewMatrixInvers;
        ModelviewMatrixInvers = new double[16];
        __gluInvertMatrixd(ModelviewMatrix_Speicher, ModelviewMatrixInvers);

        // Berechnung des Vektoers b - dieser ist die Drehachse
        bx = ModelviewMatrixInvers[0] * ax + ModelviewMatrixInvers[4] * ay +
```

```
                ModelviewMatrixInvers[8] * az;
            by = ModelviewMatrixInvers[1] * ax + ModelviewMatrixInvers[5] * ay +
                ModelviewMatrixInvers[9] * az;
            bz = ModelviewMatrixInvers[2] * ax + ModelviewMatrixInvers[6] * ay +
                ModelviewMatrixInvers[10] * az;
            // Drehen um den Dreiecksmittelpunkt indem man zuerst in diesen
            // Mittelpunkt fährt

                Gl.glTranslated(xCenter, yCenter, 0.0);
                Gl.glRotated(angle, bx, by, bz);
                Gl.glTranslated(-xCenter, -yCenter, 0.0);

            xStart = e.X;
            yStart = e.Y;
            this.SimpleOpenGlControl.Refresh();
        }
    }

    ////////////////////////////////////////////////////////////////////////////
    // Mathematik
    ////////////////////////////////////////////////////////////////////////////

    /// <summary>
    /// Die Funktion berechnet aus 2D-Mauskoordinaten (GDI-Windowskoordinaten in Pixeln)
    /// die zugehörigen 3D-OpenGL Weltkoordinaten.
    /// Wird kein Objekt, sondern der Hintergrund getroffen, wird false zurückgeliefert.
    /// Definition des Hintergrundes: Ein Pixel mit einem z-Wert > 0.9999 ( far = 1.0 )
    /// </summary>
    /// <param name="Mouse_X"></param>
    /// <param name="Mouse_Y"></param>
    /// <param name="WorldMouse_X"></param>
    /// <param name="WorldMouse_Y"></param>
    /// <param name="WorldMouse_Z"></param>
    /// <returns></returns>
    private bool MouseToWorld(int Mouse_X, int Mouse_Y, ref double WorldMouse_X,
                              ref double WorldMouse_Y, ref double WorldMouse_Z)
    {
        int[] viewport = new int[4];
        double[] ModelviewMatrix = new double[16];
        double[] ProjectionMatrix = new double[16];
        int OGL_Y;// in OpenGL-Windows-Koordinaten zeigt die y-Achse nach oben
        // Matrizen auslesen
        Gl.glGetIntegerv(Gl.GL_VIEWPORT, viewport);
        Gl.glGetDoublev(Gl.GL_MODELVIEW_MATRIX, ModelviewMatrix);
        Gl.glGetDoublev(Gl.GL_PROJECTION_MATRIX, ProjectionMatrix);
        // Richtungsumkehr der y-Achse
        OGL_Y = viewport[3] - Mouse_Y - 1;

        float[] tiefe = new float[1];
        tiefe[0] = 0.0f;
        Gl.glReadPixels(Mouse_X, OGL_Y, 1, 1, Gl.GL_DEPTH_COMPONENT, Gl.GL_FLOAT, tiefe);
        // Berechnung der Weltkoordinaten
        Glu.gluUnProject((double)Mouse_X, (double)OGL_Y, (double)tiefe[0],
         ModelviewMatrix, ProjectionMatrix, viewport, out WorldMouse_X, out WorldMouse_Y,
         out WorldMouse_Z);
        if (tiefe[0] > 0.9999) // Test: wurde der Hintergrund getroffen?
            return false;  // nicht auf Objekt geklickt
        else
            return true;
    }

    /// <summary>
    /// Die Funktion berechnet die inverse Modelviematrix.
    /// Quelle: www.mesa3D.org
    /// Mesa 3D Graphics Library: Mesa 7.1.5
    /// </summary>
    /// <param name="m"></param>
    /// <param name="invOut"></param>
    /// <returns></returns>
    private bool __gluInvertMatrixd(double[] m, double[] invOut)
    {
        double[] inv = new double[16];
        double det;
        int i;
        inv[0] = m[5] * m[10] * m[15] - m[5] * m[11] * m[14] - m[9] * m[6] * m[15]
                 + m[9] * m[7] * m[14] + m[13] * m[6] * m[11] - m[13] * m[7] * m[10];
        inv[4] = -m[4] * m[10] * m[15] + m[4] * m[11] * m[14] + m[8] * m[6] * m[15]
                 - m[8] * m[7] * m[14] - m[12] * m[6] * m[11] + m[12] * m[7] * m[10];
```

```
            inv[8] = m[4] * m[9] * m[15] - m[4] * m[11] * m[13] - m[8] * m[5] * m[15]
                     + m[8] * m[7] * m[13] + m[12] * m[5] * m[11] - m[12] * m[7] * m[9];
            inv[12] = -m[4] * m[9] * m[14] + m[4] * m[10] * m[13] + m[8] * m[5] * m[14]
                     - m[8] * m[6] * m[13] - m[12] * m[5] * m[10] + m[12] * m[6] * m[9];
            inv[1] = -m[1] * m[10] * m[15] + m[1] * m[11] * m[14] + m[9] * m[2] * m[15]
                     - m[9] * m[3] * m[14] - m[13] * m[2] * m[11] + m[13] * m[3] * m[10];
            inv[5] = m[0] * m[10] * m[15] - m[0] * m[11] * m[14] - m[8] * m[2] * m[15]
                     + m[8] * m[3] * m[14] + m[12] * m[2] * m[11] - m[12] * m[3] * m[10];
            inv[9] = -m[0] * m[9] * m[15] + m[0] * m[11] * m[13] + m[8] * m[1] * m[15]
                     - m[8] * m[3] * m[13] - m[12] * m[1] * m[11] + m[12] * m[3] * m[9];
            inv[13] = m[0] * m[9] * m[14] - m[0] * m[10] * m[13] - m[8] * m[1] * m[14]
                     + m[8] * m[2] * m[13] + m[12] * m[1] * m[10] - m[12] * m[2] * m[9];
            inv[2] = m[1] * m[6] * m[15] - m[1] * m[7] * m[14] - m[5] * m[2] * m[15]
                     + m[5] * m[3] * m[14] + m[13] * m[2] * m[7] - m[13] * m[3] * m[6];
            inv[6] = -m[0] * m[6] * m[15] + m[0] * m[7] * m[14] + m[4] * m[2] * m[15]
                     - m[4] * m[3] * m[14] - m[12] * m[2] * m[7] + m[12] * m[3] * m[6];
            inv[10] = m[0] * m[5] * m[15] - m[0] * m[7] * m[13] - m[4] * m[1] * m[15]
                     + m[4] * m[3] * m[13] + m[12] * m[1] * m[7] - m[12] * m[3] * m[5];
            inv[14] = -m[0] * m[5] * m[14] + m[0] * m[6] * m[13] + m[4] * m[1] * m[14]
                     - m[4] * m[2] * m[13] - m[12] * m[1] * m[6] + m[12] * m[2] * m[5];
            inv[3] = -m[1] * m[6] * m[11] + m[1] * m[7] * m[10] + m[5] * m[2] * m[11]
                     - m[5] * m[3] * m[10] - m[9] * m[2] * m[7] + m[9] * m[3] * m[6];
            inv[7] = m[0] * m[6] * m[11] - m[0] * m[7] * m[10] - m[4] * m[2] * m[11]
                     + m[4] * m[3] * m[10] + m[8] * m[2] * m[7] - m[8] * m[3] * m[6];
            inv[11] = -m[0] * m[5] * m[11] + m[0] * m[7] * m[9] + m[4] * m[1] * m[11]
                     - m[4] * m[3] * m[9] - m[8] * m[1] * m[7] + m[8] * m[3] * m[5];
            inv[15] = m[0] * m[5] * m[10] - m[0] * m[6] * m[9] - m[4] * m[1] * m[10]
                     + m[4] * m[2] * m[9] + m[8] * m[1] * m[6] - m[8] * m[2] * m[5];

            det = m[0] * inv[0] + m[1] * inv[4] + m[2] * inv[8] + m[3] * inv[12];
            if (det == 0)
                return false;

            det = 1.0 / det;

            for (i = 0; i < 16; i++)
                invOut[i] = inv[i] * det;

            return true;
        }

        /// <summary>
        /// Der Reset-Button.
        /// Alle Bildtransformationen werden zurückgesetzt und das Bild neu gezeichnet.
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void Resetbutton_Click(object sender, EventArgs e)
        {
            InitProjectionMatrix();
            InitModelviewMatrix();
            // Focus wieder auf das OpenGL-Window setzen
            this.SimpleOpenGlControl.Focus();
            this.SimpleOpenGlControl.Refresh();
        }
    }
}
```